Cryptographic Operator Authentication for AI Agents: Verifying the Human in the Loop

Zackary S. Parsons
Independent Researcher
ORCID: 0000-0001-7945-5128

February 2026

Abstract

As AI agents gain the ability to execute consequential actions autonomously, a critical question emerges that existing safety research largely overlooks: how does the agent verify that the human commanding it is authorized? Most AI safety work focuses on constraining the agent. This paper addresses the reverse problem: authenticating the operator to the agent.

We present a hardware-backed operator authentication protocol that repurposes cryptocurrency hardware wallets as signing devices for AI agent command authorization. The protocol uses ECDSA on secp256k1 (the Bitcoin elliptic curve) to cryptographically bind approval decisions to a specific hardware device. Key properties include: (1) signing requires physical button press and PIN entry on the device, making it immune to software-only attacks; (2) verification is performed offline by the agent using pure ECDSA key recovery, requiring no access to the signing device; (3) canonical message formatting binds signatures to specific approval records, preventing replay; (4) a USB presence monitor extends point-in-time authentication to continuous physical presence verification; and (5) the trust anchor is stored in a root-owned configuration file that the agent cannot modify.

We identify two distinct trust domains in human-AI authentication: operator-facing (proving identity to the agent) and public-facing (proving authorship to the world), and argue that these require different cryptographic mechanisms. The system has been deployed as part of Luna, an autonomous AI agent with phased autonomy infrastructure.

1 Introduction

The relationship between a human operator and an AI agent creates an authentication problem that traditional security models do not address. In conventional computing, authentication is unidirectional: a system verifies a user's identity before granting access. In the AI agent context, both directions matter. The agent needs to verify the operator (to ensure commands are legitimate), and the operator needs to verify the agent (to ensure it is behaving correctly). Current work focuses almost exclusively on the second direction.

This asymmetry creates a concrete vulnerability. Consider an autonomous AI agent with an approval queue: the agent proposes external actions (code pushes, deployments, configuration changes), and a human reviews and approves them. Without operator authentication, the approval process is vulnerable to:

- **Queue poisoning**: an attacker who gains write access to the approval queue could mark malicious actions as approved.

- **Impersonation**: in a multi-user environment, one user could approve actions beyond their authorization level.
- **Replay**: a previously valid approval could be resubmitted for a different action.
- **Coercion via prompt injection**: an adversarial input could manipulate the agent into treating unauthenticated signals as operator commands.

These are not theoretical risks. The "Codex Ralph incident" (documented by the AIM project [1]) saw an autonomous agent file a public issue on an external repository using its operator's identity, highlighting why identity boundaries in agent systems require cryptographic enforcement.

This paper makes three contributions:

1. **Authentication direction reversal**: we formalize the problem of the AI agent verifying the human operator, complementing existing work on agent identity verification [1, 2].
2. **Hardware wallet repurposing**: we demonstrate that cryptocurrency hardware wallets, designed for transaction signing, can serve as general-purpose command authorization devices for AI agents, providing hardware-backed authentication without custom hardware.
3. **Continuous physical presence**: we extend point-in-time signature verification to continuous presence verification by monitoring the signing device's USB connection state, creating a physical tether between the operator and the agent's network access.

2 Background

2.1 ECDSA and secp256k1

The Elliptic Curve Digital Signature Algorithm (ECDSA) [3] provides digital signatures using elliptic curve cryptography. The secp256k1 curve, defined by Certicom and adopted by Bitcoin [4], uses the equation $y^2 = x^3 + 7$ over a 256-bit prime field. ECDSA on secp256k1 provides 128-bit security with 32-byte private keys and 33/65-byte public keys.

A key property of ECDSA that this protocol exploits is **public key recovery**: given a message and its signature, the public key that produced the signature can be computed without prior knowledge of the signer's identity. This enables offline verification where the verifier needs only the expected public key (or derived address), not the signing device.

2.2 Bitcoin Message Signing

Bitcoin defines a message signing protocol [11] that wraps arbitrary text in a structured envelope before signing. The full serialization is: the magic prefix byte `0x18`, the string `Bitcoin Signed Message:\n`, a Bitcoin-style varint encoding of the message length, and the message bytes. This concatenation is then double-SHA256 hashed to produce the 32-byte digest that is signed. The magic prefix byte (`0x18` = 24, the length of the string "Bitcoin Signed Message:\n") prevents cross-protocol signature reuse by ensuring that Bitcoin message signatures cannot be valid transaction signatures, and vice versa [4]. The resulting signature is 65 bytes: a recovery flag byte (27–34, encoding recovery ID and key compression) followed by the 32-byte r and s values.

This protocol is natively supported by all major cryptocurrency hardware wallets, providing a ready-made signing interface for arbitrary messages without requiring custom firmware.

2.3 Hardware Wallets as Signing Devices

Cryptocurrency hardware wallets (Trezor [5], Ledger, etc.) are purpose-built devices for managing cryptographic keys. They store private keys in secure elements or isolated processors, require physical interaction (button press, touchscreen confirmation) and PIN entry for every signing operation, and never expose private keys to the host computer.

These properties make hardware wallets suitable for applications beyond cryptocurrency: any protocol that requires human-initiated, hardware-attested cryptographic signatures can leverage the existing device ecosystem.

2.4 Agent Identity Frameworks

The Binding Agent ID (BAID) framework [2] proposes cryptographic guarantees for agent identity integrity, operator legitimacy, and execution provenance. It identifies that "traditional key-based authentication is insufficient for autonomous agents, as possession of a signing key guarantees neither the integrity of the executing code nor the authenticity of the operator." The Agent Identity Management (AIM) platform [1] provides cryptographic identity and access control for AI agents.

Both frameworks focus primarily on the agent's identity as perceived by external systems. The operator authentication problem, where the agent must verify commands from the human, receives less attention.

Ledger's CTO recently proposed an "Agents Propose, Humans Sign" paradigm [6] for AI agent cryptocurrency transactions, recognizing that "a compromised agent could initiate irreversible on-chain actions." This is conceptually similar to our approval queue model, but applied to financial transactions rather than general agent command authorization.

3 Threat Model

3.1 System Model

We consider an AI agent system with the following components:

- An **agent** that runs on infrastructure controlled by the operator and can propose external actions.
- An **approval queue** where proposed actions are recorded and await authorization.
- An **operator** who reviews proposed actions and decides whether to approve them.
- A **signing device** (hardware wallet) held by the operator.
- A **verifier** that runs within the agent's execution environment.

3.2 Attacker Capabilities

We consider three attacker classes:

Software-only attacker: has arbitrary code execution on the agent's host machine but no physical access to the signing device. This models a compromised agent session (via prompt injection, model misalignment, or software vulnerability).

Network attacker: can intercept and modify network traffic between the agent and external services but cannot access the host machine. This models man-in-the-middle attacks on the approval channel.

Insider with limited access: has some access to the agent system (e.g., can write to the approval queue) but does not possess the signing device. This models multi-user environments or compromised secondary accounts.

3.3 Trust Assumptions

- The signing device correctly implements ECDSA on secp256k1 and does not leak private keys.
- The operator's PIN and physical possession of the device have not been compromised.
- The trust anchor (expected signing address) is stored in a location the agent cannot modify (root-owned file).
- The ECDSA problem (computing private keys from public keys) is computationally infeasible.

3.4 Security Goals

1. **Authenticity**: only the holder of the signing device can produce valid approvals.
2. **Non-repudiation**: a valid approval cryptographically proves that the signing device was used.
3. **Binding**: each signature is bound to a specific approval record and cannot be reused for a different action.
4. **Freshness**: the approval includes a timestamp that prevents indefinite replay.
5. **Tamper resistance**: the agent cannot forge, modify, or bypass the authentication check.

4 Protocol Design

4.1 Enrollment

During initial setup, the operator derives a signing key from the hardware wallet using a deterministic path (BIP-44 [10]: `m/44'/0'/0'/0/0`). The corresponding Bitcoin P2PKH address is computed and stored as the trust anchor in a root-owned configuration file:

```
/etc/luna/trezor-trusted.conf
TRUSTED_ADDRESS=<base58check P2PKH address>
BIP44_PATH=m/44'/0'/0'/0/0
DEVICE=<device model>
```

This file is owned by root with permissions 0644 (readable by the agent, writable only by the system administrator). The agent reads the trusted address at runtime but cannot modify it.

4.2 Canonical Message Format

Each approval request in the queue has a unique identifier, an action type, and a timestamp. The signing protocol constructs a canonical message that binds the signature to a specific approval record:

```
luna-approve:<approval_id>:<action_type>:<requested_at>
```

For example: `luna-approve:42:git_push:2026-02-13T15:30:00Z`
The canonical format ensures that:

- Signatures cannot be replayed for different approval IDs (binding).
- The action type is included in the signed data (preventing type confusion).
- The timestamp provides freshness (approvals for old requests can be flagged).

4.3 Signing Protocol

When the operator decides to approve an action:

1. The approval script retrieves the pending approval record from the database.
2. The canonical message is constructed from the record's fields.
3. The script invokes the hardware wallet's message signing command with the canonical message.
4. The device displays the message and requires PIN entry and physical button confirmation.
5. The device returns a 65-byte Bitcoin message signature.
6. The script extracts the signing address from the signature using ECDSA key recovery.
7. The script verifies that the recovered address matches the trusted address.
8. If verification succeeds, the signature and address are stored in the approval record's `resolved_by` field using a structured format: `trezor:<address>:<base64_signature>`.
9. The approval status is updated to `approved`.

4.4 Offline Verification

The verification algorithm runs without access to the signing device:

1. Parse the stored signature from the `resolved_by` field.
2. Reconstruct the canonical message from the approval record's fields.
3. Compute the Bitcoin message hash: `SHA256(SHA256(0x18 + "Bitcoin Signed Message:\n" + varint(len(msg)) + msg))`.
4. Recover the public key from the signature using the recovery flag byte and the (r, s) values.
5. Derive the P2PKH address: `Base58Check(0x00 + RIPEMD160(SHA256(compressed_pubkey)))`.
6. Compare the derived address against the trusted address from the root-owned configuration.
7. Accept if and only if the addresses match.

Steps 3–6 are pure mathematical operations requiring only the `ecdsa` library (or equivalent) and standard hash functions. No network access, no device communication, and no external services are needed.

4.5 Continuous Physical Presence

Point-in-time signature verification proves the operator authorized a specific action at a specific time. It does not prove continuous presence. An operator could sign an approval, then leave, while the agent continues to operate with network access.

To address this, a USB presence monitor polls the system's USB device tree at short intervals (5 seconds) for the signing device. When the device is detected:

- SSH keys are loaded into the agent with per-use confirmation (requiring biometric or PIN for each use).
- A state file records the device as present.

When the device is removed:

- All SSH keys are flushed from the agent.
- The state file records the device as absent.

This creates a physical tether: the operator must have the signing device physically connected for the agent to have network access. Removing the device immediately revokes all SSH keys, cutting the agent's ability to interact with remote systems.

The round-trip time (device removal to key flush to device reconnection to key reload) is approximately 20 seconds in practice.

4.6 Trust Anchor Design

The trust anchor must satisfy two properties: the agent can read it (to perform verification), and the agent cannot modify it (to prevent trust anchor substitution).

This is achieved through OS-level file ownership. The configuration file containing the trusted address is owned by root and has permissions 0644. The agent process runs as a non-root user. Even if the agent is compromised, it cannot replace the trusted address with an attacker-controlled address because it lacks write permission to the file.

For additional defense in depth, the trust anchor can also be embedded in the signed integrity manifest that is verified before each autonomous session. Modifying the manifest requires the operator's cryptographic signature, creating a circular dependency that prevents unilateral changes by the agent.

5 Security Analysis

5.1 Resistance to Forgery

An attacker who controls the agent's host machine but does not possess the hardware wallet cannot produce a valid signature. The private key never leaves the hardware device. Forging an ECDSA signature on secp256k1 requires solving the elliptic curve discrete logarithm problem, which is computationally infeasible with current technology (128-bit security level).

5.2 Resistance to Replay

The canonical message format includes the approval ID, action type, and timestamp. A signature for approval #42 cannot be used to approve action #43 because the approval ID in the reconstructed canonical message will not match. An attacker who intercepts a valid signature can only "replay" it for the exact same approval record, which is a no-op (the action is already approved).

Cross-approval replay is prevented because each approval record has a unique combination of ID, type, and timestamp.

5.3 Resistance to Agent Manipulation

The agent cannot manipulate the verification process because:

- The trust anchor is in a root-owned file (agent cannot substitute a different address).
- The verification algorithm is deterministic (given the same inputs, it always produces the same result).
- The canonical message is reconstructed from the database record, not from agent-supplied parameters (preventing the agent from constructing a message that would match a different signature).

A compromised agent could attempt to:

- **Modify the database record** after signing but before verification. Mitigation: verification reconstructs the canonical message from the record; any modification invalidates the signature.
- **Replace the verification script**. Mitigation: the script is covered by the integrity manifest, which is verified before each autonomous session using a separate, root-owned verifier.
- **Skip verification entirely**. Mitigation: the approval execution path requires a valid signature; skipping verification means the `resolved_by` field remains empty, and the action is not marked as approved.

5.4 Key Compromise Scenarios

Signing device lost or stolen: the attacker needs the PIN to use the device. After several incorrect PIN attempts, the device wipes its keys. The operator recovers by restoring from the device's seed phrase (which deterministically regenerates the same keys) and updating the trust anchor if the device model changes.

Seed phrase compromised: the attacker can derive the private key and forge signatures without the physical device. This is the highest-impact compromise. Mitigation: seed phrases should be stored physically (not digitally) in a secure location. Detection: if forged approvals appear without the operator's knowledge, the mismatch between operator awareness and approval records is an indicator of compromise.

Host machine compromised: the attacker cannot forge signatures (the key is on the device), but could potentially intercept the canonical message displayed on the device screen and present a different message to the operator. Mitigation: the operator should verify the message content on the device's display, not on the host screen.

Approach	Authentication	Hardware-backed	Offline verify	Physical presence
Password/HMAC	Shared secret	No	Yes	No
TOTP/HOTP	Time/counter OTP	Optional (Yubikey)	No	No
WebAuthn/FIDO2	Challenge-response	Yes	No	No
PGP/GPG signing	Asymmetric signature	Optional (smartcard)	Yes	No
HW wallet (this work)	**ECDSA msg signing**	**Yes (always)**	**Yes**	**Yes (USB)**

Table 1: Comparison of operator authentication approaches.

5.5 Comparison with Alternative Approaches

Hardware wallet authentication uniquely combines all four properties. WebAuthn comes closest but requires a relying party server and does not support offline verification or continuous presence monitoring. PGP signing supports offline verification but does not enforce hardware-backed key storage and lacks a presence mechanism.

6 Two Trust Domains

A key insight from deploying this system is that human-AI authentication involves two fundamentally different trust relationships that should not share cryptographic material.

6.1 Operator-Facing Authentication (AI-Facing)

This is the domain addressed by this paper: proving the operator's identity to the AI agent. The audience is the agent itself. The verification happens in the agent's execution environment. The key properties required are: hardware-backed signing, offline verification, canonical message binding, and physical presence.

The hardware wallet is the appropriate tool because it provides hardware attestation without requiring network connectivity, and the Bitcoin message signing protocol is already implemented and battle-tested.

6.2 Public-Facing Authentication (World-Facing)

This domain involves proving authorship or identity to the broader world: signing publications, code commits, or communications. The audience is the public or a community of peers. The key properties required are: wide ecosystem support, key server infrastructure, established trust models (web of trust or certificate authorities), and long-term verifiability.

PGP/GPG signing is the appropriate tool for this domain because it has decades of ecosystem support, established key distribution infrastructure, and widespread tooling.

6.3 Why Separation Matters

Using the same key for both domains creates unnecessary risk coupling. If the operator-facing key is compromised, it should not affect the operator's public identity. If the public-facing key needs rotation (e.g., moving to a stronger algorithm), it should not require reconfiguring the agent's trust anchor.

The separation also reflects a conceptual difference: operator-facing authentication is a private, machine-to-human protocol. Public-facing authentication is a public, human-to-world protocol. Different audiences, different threat models, different keys.

7 Implementation

The system has been deployed as part of Luna, an autonomous AI agent described in a companion paper [7].

7.1 Signing Component

The signing wrapper is a shell script that invokes the hardware wallet's CLI tool to sign messages. It supports two modes: raw message signing, and approval signing where the canonical message is automatically constructed from the database record. The script verifies the signature locally before storing it, providing immediate feedback if the wrong device or key path is used.

7.2 Offline Verifier

The verifier is a Python script using the `ecdsa` library for pure-Python ECDSA key recovery on secp256k1. It accepts a message and a base64-encoded 65-byte Bitcoin message signature, recovers the public key, derives the P2PKH address, and compares against the trusted address.

The verifier has been validated against real signatures produced by a Trezor Safe 5. For production use, a more battle-tested library (such as `python-secp256k1`, which wraps Bitcoin Core's `libsecp256k1`) is recommended.

7.3 Presence Monitor

The USB presence monitor is a shell script that polls the system's USB device tree for the hardware wallet's vendor and product ID. On detection, SSH keys are loaded with per-use confirmation (requiring biometric verification for each SSH operation). On removal, all keys are flushed. The monitor runs as a scheduled background job with a 5-second polling interval.

7.4 Approval Workflow

The complete approval workflow integrates all components:

1. The agent records an approval request in the database during an autonomous session.
2. The operator is notified via push notification.
3. The operator runs the approval script, which displays the pending request.
4. The script constructs the canonical message and invokes the hardware wallet.
5. The hardware wallet displays the message on its screen for verification.
6. The operator enters their PIN and presses the physical confirm button.
7. The script verifies the signature offline against the trusted address.
8. The signature is stored in the database.
9. In a subsequent session, the agent reads the approved record and executes the action.

8 Discussion

8.1 Generalization Beyond Cryptocurrency Wallets

The protocol described here uses Bitcoin message signing, but the underlying principle is more general: any hardware device that (a) stores private keys securely, (b) requires physical interaction for signing, and (c) supports arbitrary message signing can serve as an operator authentication device. This includes FIDO2 security keys (with extensions for arbitrary data signing), smart cards with PKCS#11 interfaces, and future hardware specifically designed for AI agent authentication.

The advantage of cryptocurrency hardware wallets is availability. They are commodity devices, widely distributed, with mature firmware and established security auditing. No custom hardware is required.

8.2 Scaling to Multiple Operators

The current design supports a single operator with a single trusted address. Extending to multiple operators requires a trust anchor that maps operator identities to their respective addresses, with per-operator authorization levels. The verification logic generalizes naturally: recover the signing address, look it up in the authorized operators table, and check whether the operator has sufficient privileges for the requested action type.

8.3 Relationship to Agentic Wallets

Coinbase's Agentic Wallets [8] and similar systems give AI agents their own cryptocurrency wallets, enabling autonomous financial transactions. This is the inverse of our approach: we use a human's wallet to authenticate commands to an agent, while agentic wallets give agents their own financial agency.

Both approaches may coexist in a complete system. The agent could have its own wallet for routine transactions (bounded by spending limits), while consequential actions require operator authentication via the hardware wallet protocol described here.

8.4 USB Presence Limitations

The continuous presence mechanism relies on USB device enumeration to detect the hardware wallet. This is not a cryptographic guarantee: a sufficiently privileged attacker with physical access to the host could spoof USB vendor/product IDs to simulate device presence without the actual hardware wallet. The presence monitor is therefore a defense-in-depth measure, not a security boundary. The cryptographic signature remains the hard authentication factor; USB presence adds a soft revocation layer that raises the bar for sustained unauthorized access but should not be treated as equivalent to signature verification.

The USB monitor is also platform-specific (the current implementation targets macOS using `system_profiler`). Porting to Linux (`/sys/bus/usb/devices/`) or FreeBSD (`usbconfig`) is straightforward but has not been validated.

8.5 Trust Anchor Scope

The root-owned trust anchor file prevents modification by the agent process, but an attacker who achieves root privilege escalation can modify the file. This is a deliberate design choice: the trust anchor is not intended to defend against a root-level attacker (who already controls the machine), but against the agent process itself. The threat model (Section 3.2) explicitly scopes the software-only

attacker as having arbitrary code execution but not root privilege. Defense against a root-level attacker requires additional measures (hardware security modules, remote attestation, or air-gapped trust anchor storage) that are outside the scope of this protocol.

8.6 Post-Quantum Considerations

ECDSA on secp256k1 relies on the hardness of the elliptic curve discrete logarithm problem, which is vulnerable to Shor's algorithm on a sufficiently large quantum computer [12]. If fault-tolerant quantum computers become practical, the protocol's signing and verification components would need migration to post-quantum signature schemes (e.g., CRYSTALS-Dilithium, SPHINCS+). The protocol's modular design makes this migration feasible: the trust anchor stores a derived address, and the verification algorithm is isolated in a single script. Replacing ECDSA with a post-quantum scheme requires updating (1) the hardware wallet firmware, (2) the address derivation function, and (3) the verification script, without changes to the approval queue, canonical message format, or workflow logic. Current cryptocurrency hardware wallets do not support post-quantum signing, so this migration depends on device ecosystem readiness.

8.7 Other Limitations

The protocol requires the operator to have physical access to the hardware wallet for every approval, which may be inconvenient for high-frequency approval workflows. The offline verifier uses the `ecdsa` Python library, which is less audited than production cryptographic libraries. The system has been validated with a single hardware wallet model (Trezor Safe 5) and has not been tested across the full range of devices that support Bitcoin message signing.

9 Related Work

Nakamoto's Bitcoin protocol [4] established ECDSA on secp256k1 as a practical digital signature scheme with public key recovery, enabling the message signing protocol this work builds on. Johnson et al. [3] provide the formal specification of ECDSA. The BIP-44 standard [10] defines the hierarchical deterministic key derivation paths used for enrollment, and BIP-137 [11] specifies the message signing format.

The W3C Web Authentication standard (WebAuthn) [9] defines a protocol for hardware-backed authentication using public key credentials. WebAuthn is designed for web authentication flows and requires a relying party server for challenge generation and verification. Our protocol does not require a server, enabling fully offline verification.

The Binding Agent ID framework [2] formalizes cryptographic agent identity but focuses on the agent's identity as perceived by external systems. Our work addresses the complementary direction: the operator's identity as verified by the agent.

OpenA2A's Agent Identity Management platform [1] provides agent-centric identity and access control. Our approach could integrate with AIM as the operator authentication layer, providing the "operator legitimacy" guarantee that AIM identifies as a requirement.

Ledger's "Agents Propose, Humans Sign" paradigm [6] is the closest industry parallel to our approach. It applies the same principle (hardware-attested human authorization for agent actions) to cryptocurrency transactions. Our work generalizes this to arbitrary agent command authorization and adds continuous physical presence verification.

10 Conclusion

We have presented a hardware-backed operator authentication protocol for AI agents that reverses the conventional authentication direction: instead of constraining the agent, we verify the human. By repurposing commodity cryptocurrency hardware wallets, the protocol provides hardware-attested signing, offline verification, replay resistance through canonical message binding, and continuous physical presence monitoring, all without custom hardware or network-dependent verification infrastructure.

The identification of two distinct trust domains (operator-facing and public-facing) provides a framework for reasoning about cryptographic identity in human-AI systems. Different audiences and different threat models warrant different keys and different protocols.

As AI agents gain increasing autonomy, the question "who authorized this action?" becomes as important as "should this action be allowed?" The protocol presented here provides a cryptographic answer that is grounded in deployed hardware, validated mathematical properties, and practical operational experience.

References

[1] OpenA2A, "AIM: Agent Identity Management," https://opena2a.org, 2025.

[2] Z. Lin et al., "Binding Agent ID: Unleashing the Power of AI Agents with Accountability and Credibility," arXiv:2512.17538, December 2025.

[3] D. Johnson, A. Menezes, and S. Vanstone, "The Elliptic Curve Digital Signature Algorithm (ECDSA)," *International Journal of Information Security*, vol. 1, pp. 36–63, 2001.

[4] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008.

[5] SatoshiLabs, "Trezor Hardware Wallet," https://trezor.io.

[6] Ledger, "Agents Propose, Humans Sign: Securing AI Agent Transactions," Crowdfund Insider, February 2026.

[7] Z. S. Parsons, "Phased Autonomy: Infrastructure for Self-Directed AI Agents," preprint, 2026.

[8] Coinbase, "Introducing Agentic Wallets," https://developer.coinbase.com, 2026.

[9] W3C, "Web Authentication: An API for accessing Public Key Credentials," W3C Recommendation, 2021.

[10] M. Palatinus, P. Rusnak, A. Voisine, and S. Bowe, "BIP-44: Multi-Account Hierarchy for Deterministic Wallets," Bitcoin Improvement Proposals, 2014.

[11] C. Marr, "BIP-137: Signatures of Messages using Private Keys," Bitcoin Improvement Proposals, 2017.

[12] P. W. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer," *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1484–1509, 1997.